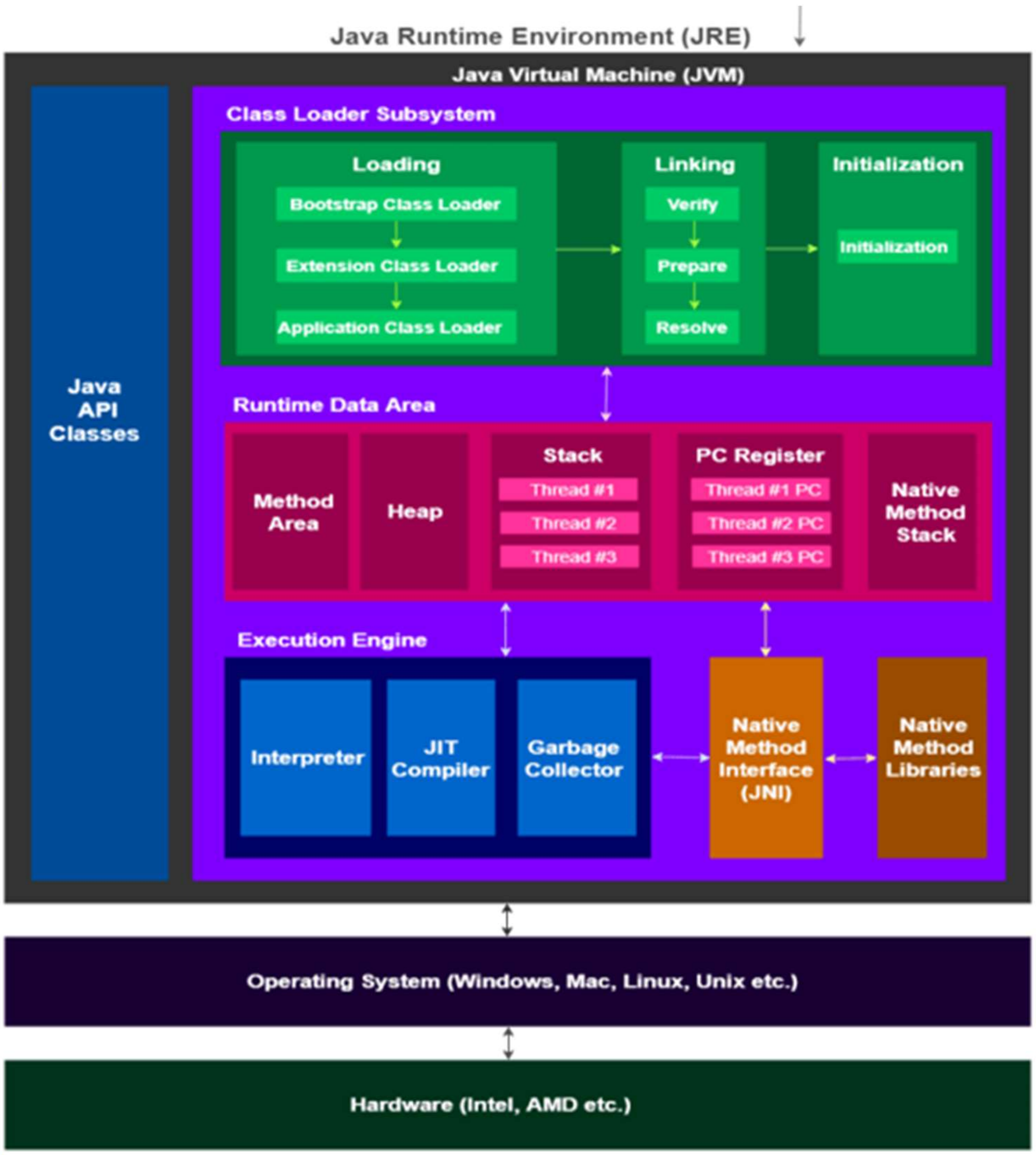
Java PerformanceTuning

Java Memory Model

Java Runtime Environment

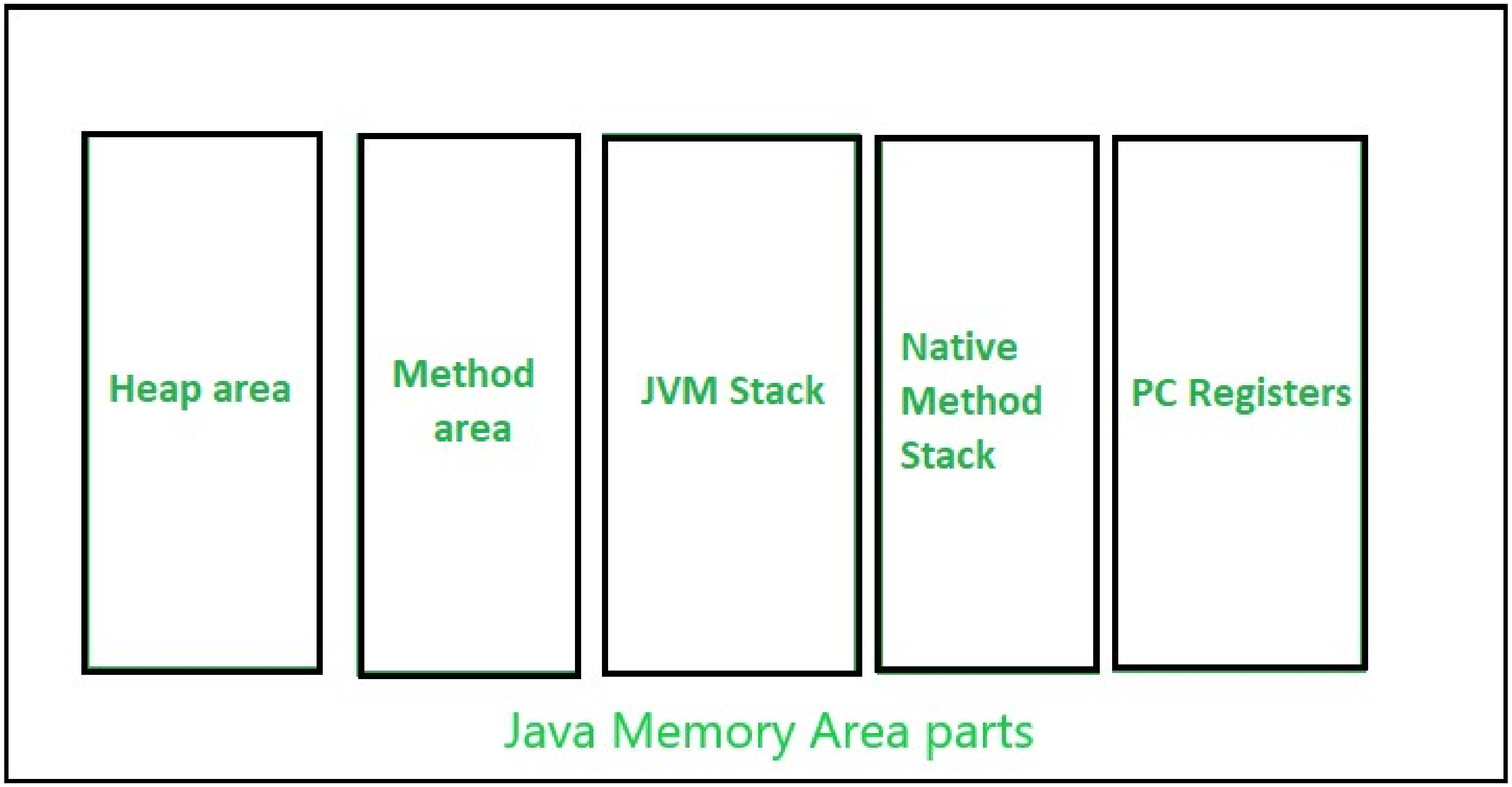


JVM Architecture
(Image: PlatformEngineer.com)

Memory model

- The Java memory model specifies how the JVM works with the computer's memory (RAM).
- The Java virtual machine is a model of a whole computer so this model naturally includes a memory model
- Divided into
  - Heap
  - Method area
  - Stack area
  - …..etc

Memory model



Memory model

Heap:

- shared runtime data area and stores the actual object in a memory
- JVM provides the user control to initialize or vary the size of heap as per the requirement.
- When a new keyword is used, object is assigned a space in heap, but the reference of the same exists onto the stack.

Stack:

- A stack is created at the same time when a thread is created and is used to store data and partial results which will be needed while returning value for method and performing dynamic linking.
- Stacks can either be of fixed or dynamic size. The size of a stack can be chosen independently when it is created

Memory model

Method area:

- memory block that stores the class code, variable code(static variable, runtime constant), method code, and the constructor of a Java program.
- It stores class-level data of every class such as the runtime constant pool, field and method data, the code for methods

Native Method stacks:

- Also called as C stacks, native method stacks are not written in Java language
- This memory is allocated for each thread when its created. And it can be of fixed or dynamic nature

Memory model

Program Counter:

- Each JVM thread which carries out the task of a specific method has a program counter register associated with it
- The non native method has a PC which stores the address of the available JVM instruction

Cache Memory

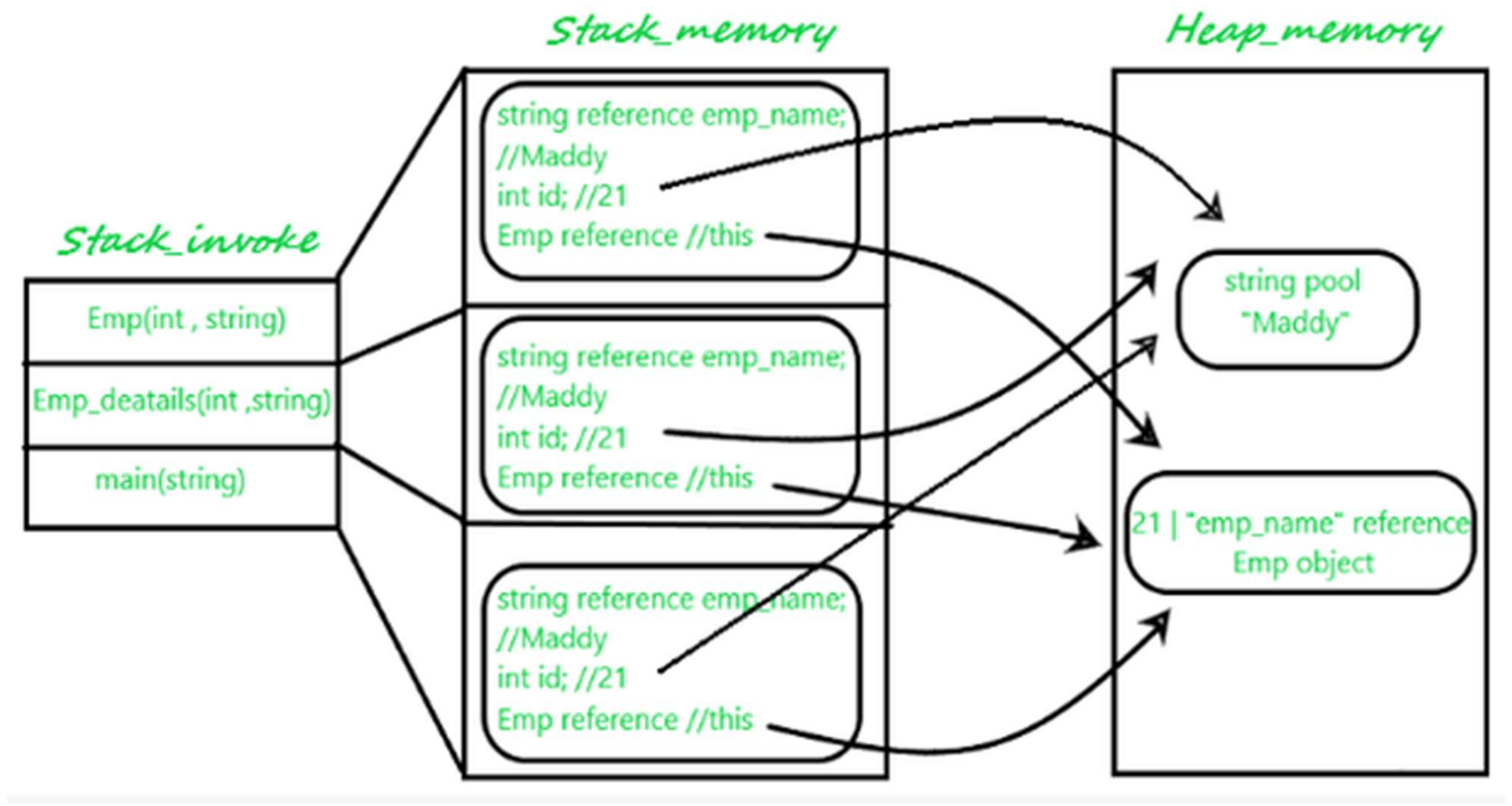
- This includes Code Cache
- Stores compiled code (i.e. native code) generated by JIT compiler, JVM internal structures etc
- When Code Cache exceeds a threshold, it gets flushed

Memory model - sample

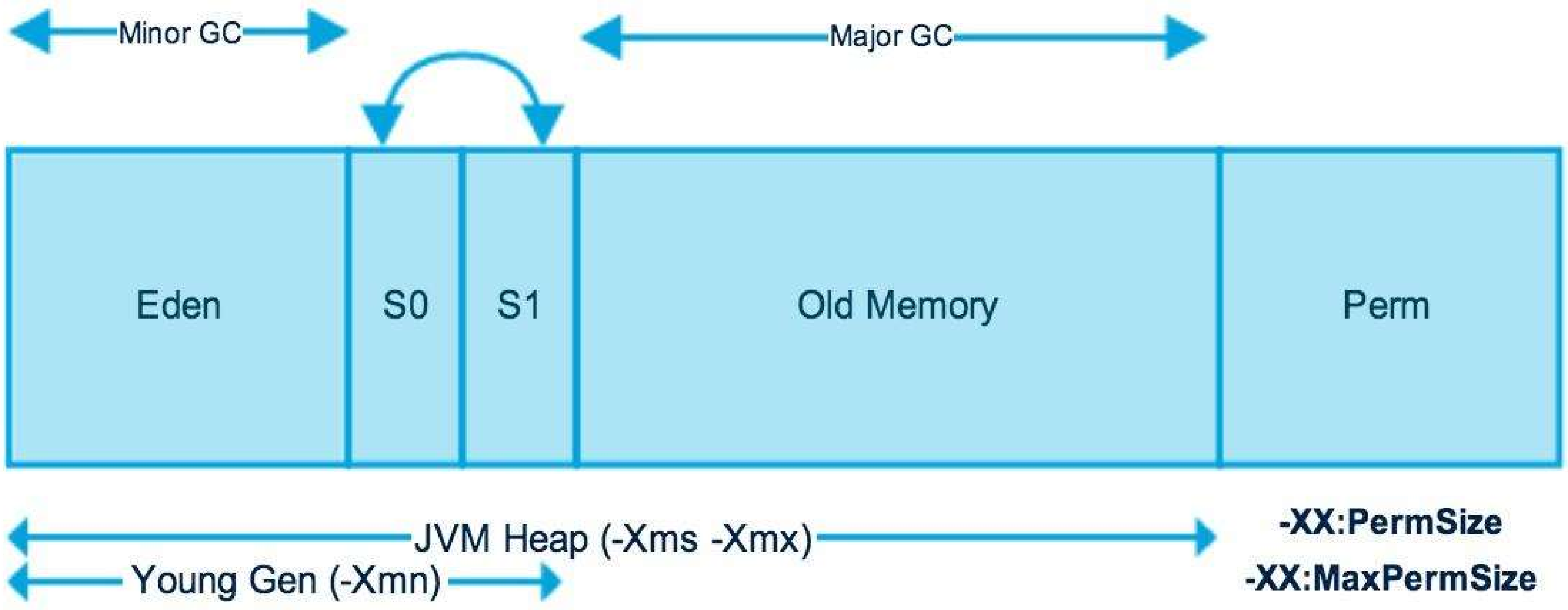
```
class Emp {
    int id;
    String emp_name;

    public Emp(int id, String emp_name) {
        this.id = id;
        this.emp_name = emp_name;
    }
}

public class Emp_detail {
    private static Emp Emp_detail(int id, String
emp_name) {
        return new Emp(id, emp_name);
    }
```

```
    public static void main(String[] args) {
        int id = 21;
        String name = "Maddy";
        Emp person = null;
        person  = Emp_detail(id, name);
    }
}
```

Memory model - sample



Heap Memory


Minor GC
Major GC
Eden
S0
S1
Old Memory
Perm
JVM Heap (-Xms -Xmx)
Young Gen (-Xmn)
-XX:PermSize
-XX:MaxPermSize

Garbage Collection

- The JVM heap is physically divided into two parts: *nursery* (*young generation*) and *old space* (or *old generation*)
- The nursery is a part of the heap reserved for the allocation of new objects.
- When the nursery becomes full, all the objects that have lived long enough in the nursery are promoted (moved) to the old space, thus freeing up the nursery for more object allocation
- This garbage collection is called **Minor GC**.
- The nursery is divided into three parts – **Eden Memory** and two **Survivor Memory** spaces
- When old space is full, **Major GC** happens

Garbage Collection

- Most of the newly created objects are located in the Eden Memory space
- When Eden space is filled with objects, Minor GC is performed and all the survivor objects are moved to one of the survivor spaces
- Minor GC also checks the survivor objects and moves them to the other survivor space.
- So at a time, one of the survivor space is always empty
- Objects that have survived many cycles of GC, are moved to the old generation memory space.
- Usually, it is done by setting a threshold for the age of the nursery objects before they become eligible to promote to the old generation

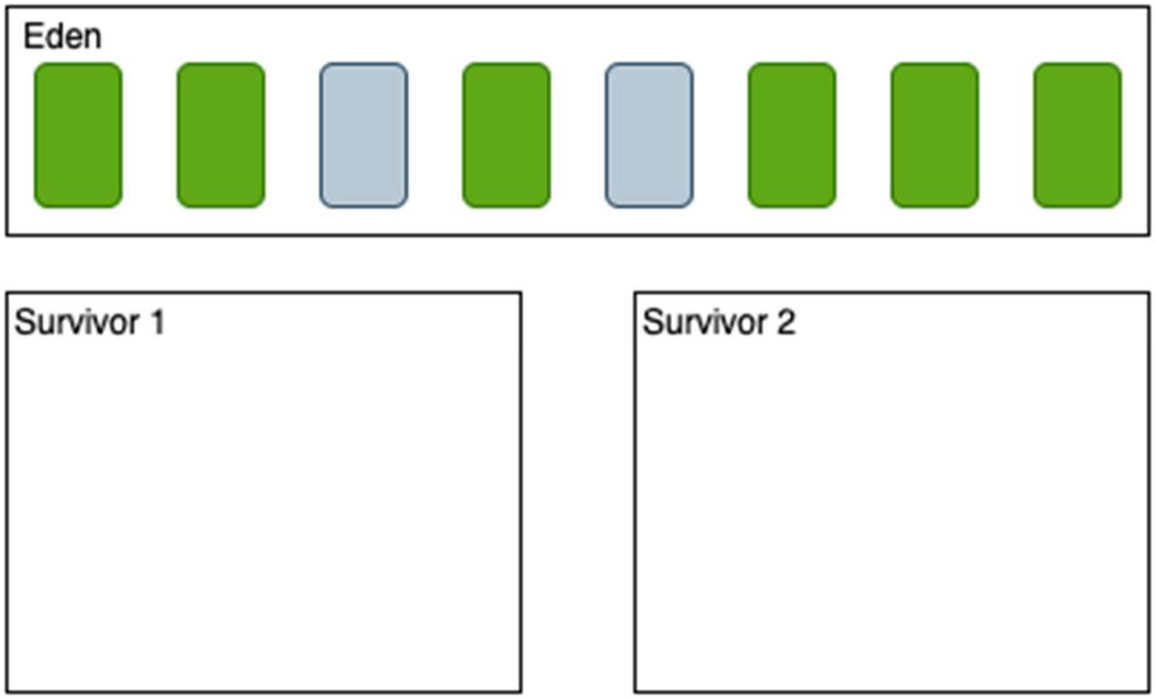
SutvivorRatio

- The SurvivorRatio parameter controls the size of the two survivor space
- **-XX:SurvivorRatio=6** sets the ratio between each survivor space and eden to be 1:6

- If survivor spaces are too small, copying collection overflows directly into the old generation
- If survivor spaces are too large, they will be empty
- At each GC, the JVM determines the number of times an object can be copied before it is tenured, called the tenure threshold

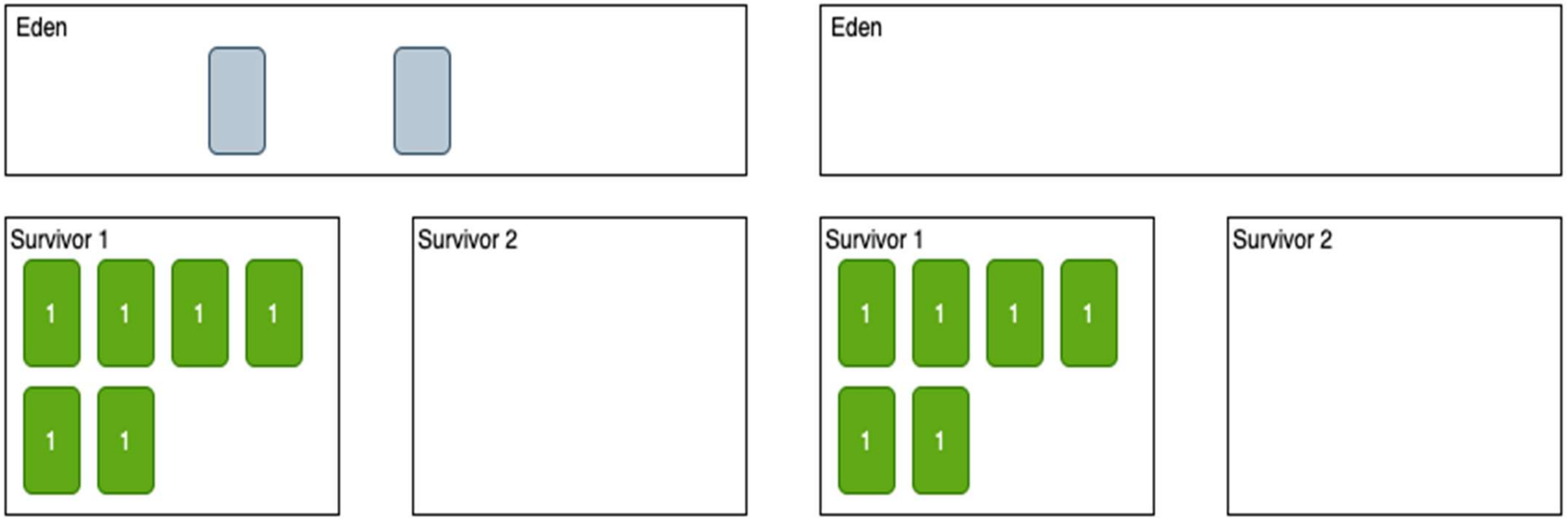
Garbage Collection

- All new objects are created in Eden space
- When the given space is full, the application tries to create another object and JVM tries to allocate something on the Eden but the allocation fails
- That actually causes minor GC



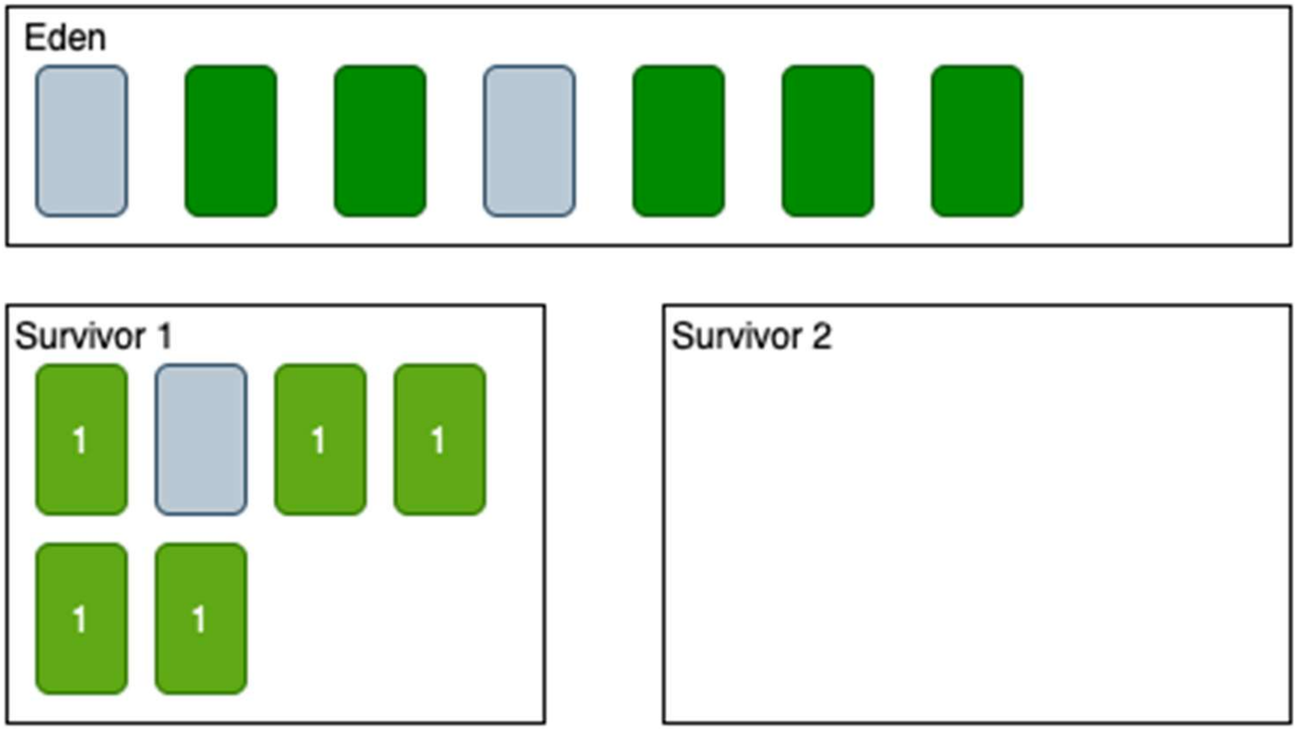
Garbage Collection

- After the first minor GC, all live objects will be moved to Survivor 1 with the age is 1 and the dead objects will be deleted



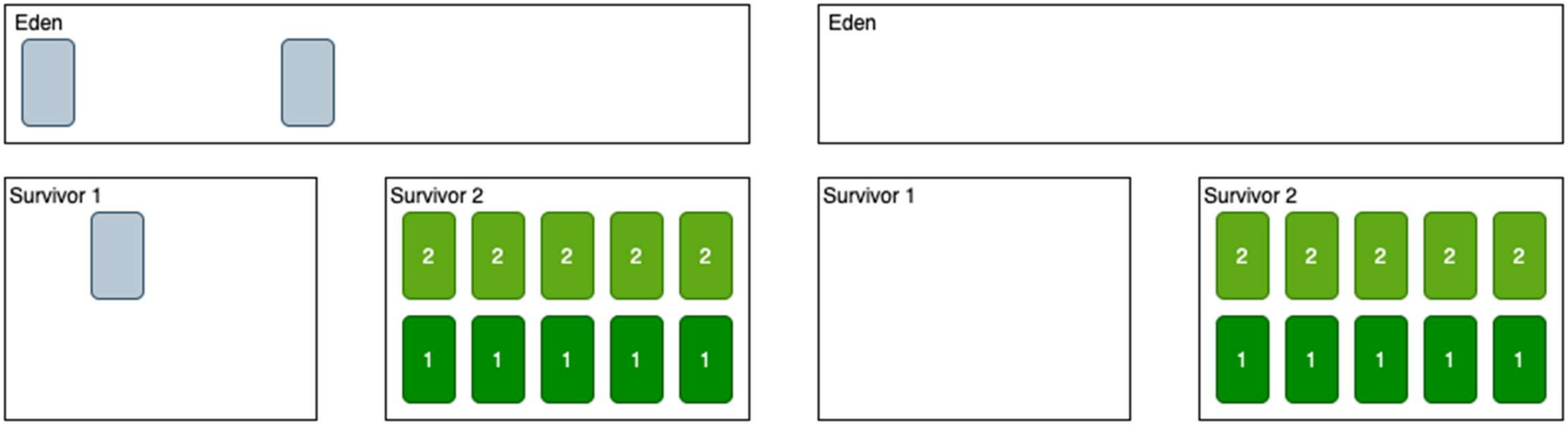
Garbage Collection

- new objects get allocated in Eden space again. There are some objects that become unreachable on both Eden space and Survivor 1



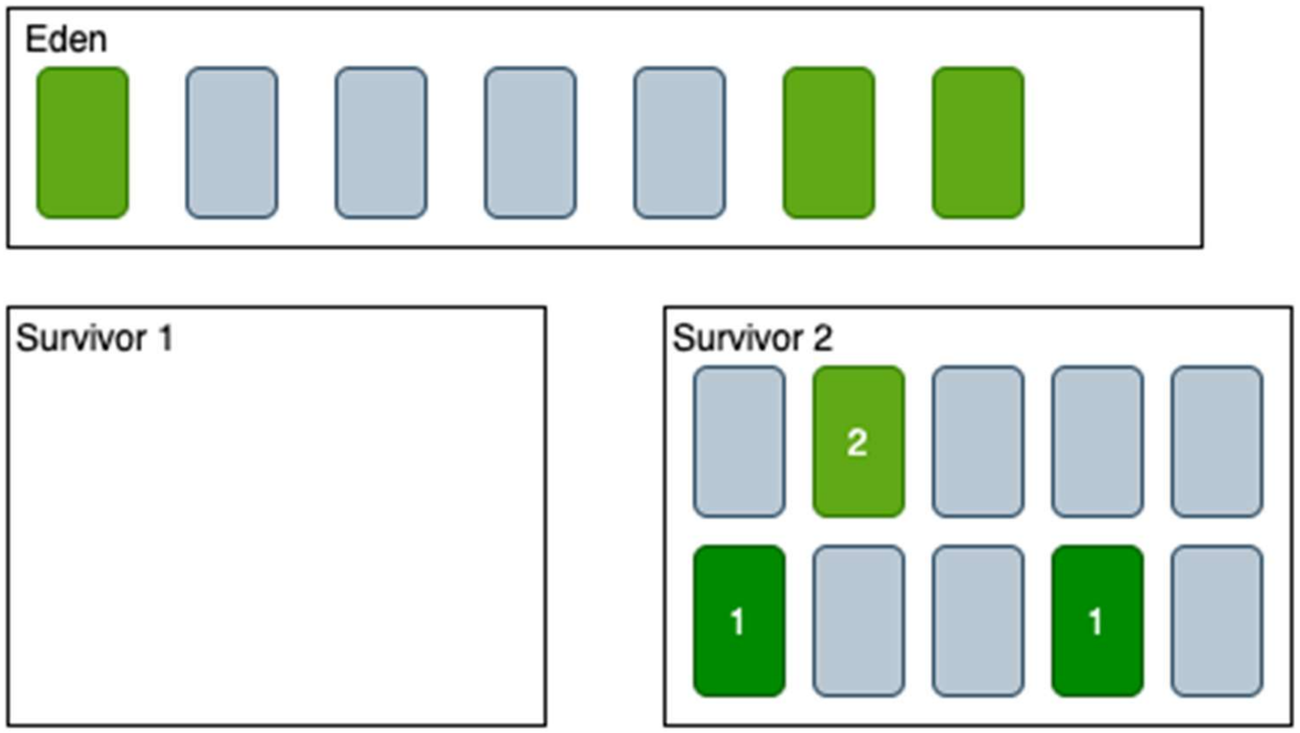
Garbage Collection

- After the second Minor GC, all alive objects will be moved to Survivor 2 (from both Eden with age 1 and Survivor 1 with age 2) and the dead object will be deleted



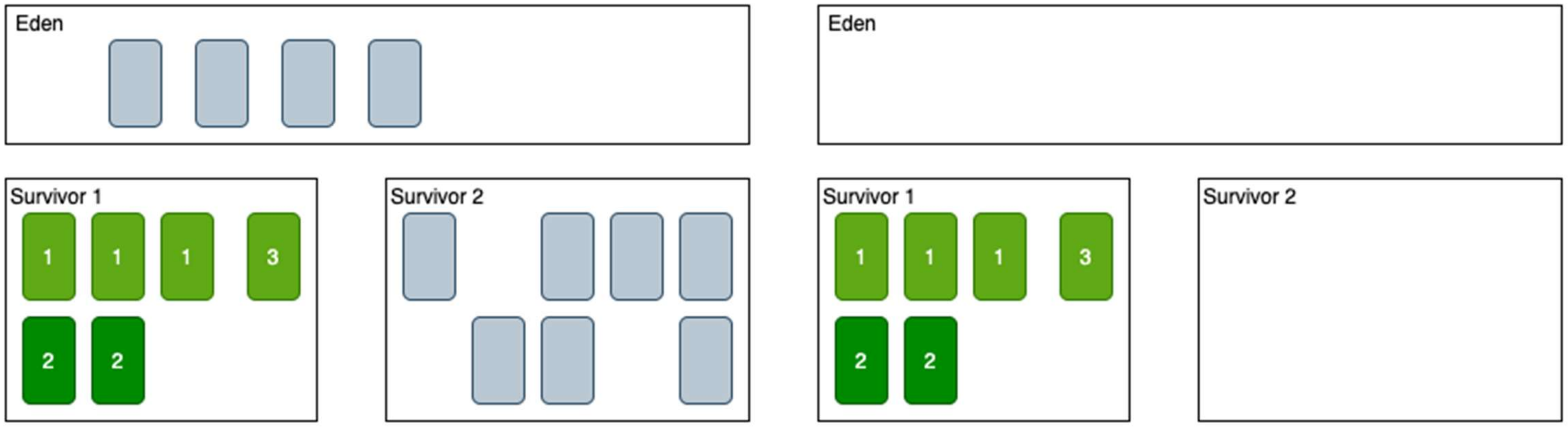
Garbage Collection

- new objects are allocated on Eden space, after a few moments some objects are unreachable from both Eden and Survivor 2



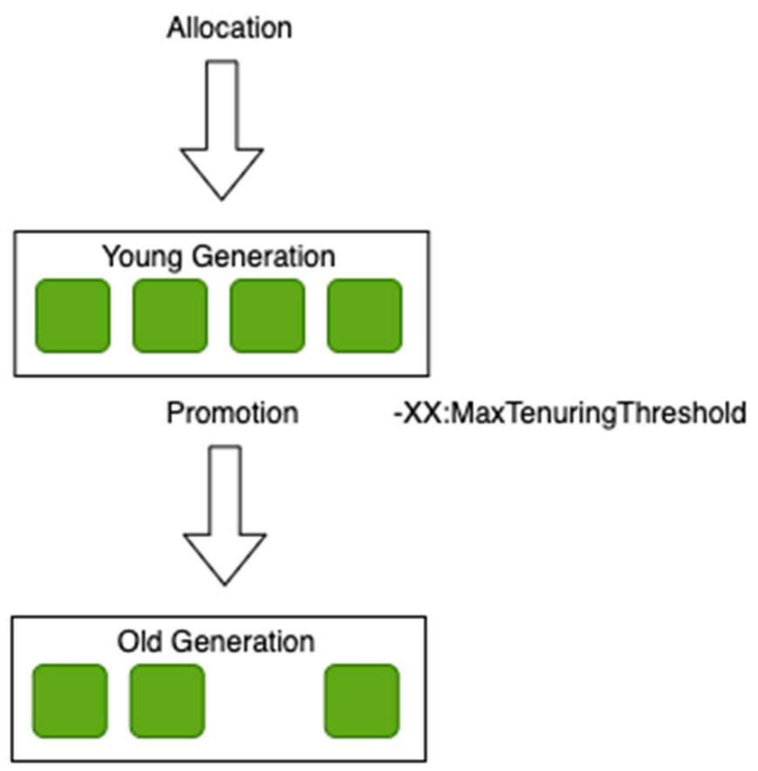
Garbage Collection

- After the third minor GC, all live objects will be move from both Eden and Survivor 2 to Survivor 1 with age increase and dead objects will be deleted



Garbage Collection

- An object that is living longer in Survivor will be promoted to the old generation (Tuner) if the age is greater than -XX:MaxTenuringThreshold



MaxTenuringThreshold

- When tuning the Java garbage collector (GC), 'survivor spaces' can be configured to "age" new objects.

- The Java command line parameter -XX:MaxTenuringThreshold specifies for how many minor GC cycles an object will stay in the survivor spaces until it finally gets tenured into the old space

  Default value is 15.

  If a value greater than 15 is set, this now specifies that objects should never tenure and leads heap fragmentation

- A fragmented heap cannot accommodate as many objects as a compacted heap

PermGen / Metaspace

- Permanent Generation or “Perm Gen” contains the application metadata required by the JVM to describe the classes and methods used in the application.
- Perm Gen is populated by JVM at runtime based on the classes used by the application.
- Perm Gen also contains Java SE library classes and methods
- With Java 8, Perm Gen is replaced by **Metaspace** which is not part of the heap
- Most allocations of the class metadata are now allocated out of native memory.

PermGen / Metaspace (contd)

- Metaspace by default auto increases its size (up to what the underlying OS provides), while Perm Gen always has fixed maximum size.
- The theme behind the Metaspace is that the lifetime of classes and their metadata matches the lifetime of the class loaders.
- That is, as long as the classloader is alive, the metadata remains alive in the Metaspace and can’t be freed.

Other structures

Method Area

- Method Area used to store class structure, static variables and code for methods and constructors
- Although the method area is logically part of the heap, simple implementations may choose not to either garbage collect or compact it

Memory Pool

- Memory Pools are created by JVM memory managers to create a pool of immutable objects.
- Memory Pool can belong to Heap or Perm Gen, depending on JVM memory manager implementation.

Other structures

Runtime Constant Pool

- Runtime constant pool is a per-class runtime representation of a constant pool in a class. It contains class runtime constants and static methods.
- Runtime constant pool is part of the method area.

Setting memory sizes

-Xms	sets initial heap size
-Xmx	sets maximum heap size
-XX:PermSize	sets initial size of permgen
-XX:MaxPermSize	sets maximum size of permgen
-XX:NewSize	*sets initial size of young generation*
-XX:MaxNewSize	*sets max size of young generation*
-XX:SurvivorRatio	*sets ratio of eaden space to survivor space*
-XX:NewRatio	*sets ratio old space to new space*
-XX:MetaspaceSize	*sets initial size of metaspace*
-XX:MetaspaceSize	sets max size of metaspace
-Xss	sets the size of thread stack for each thread

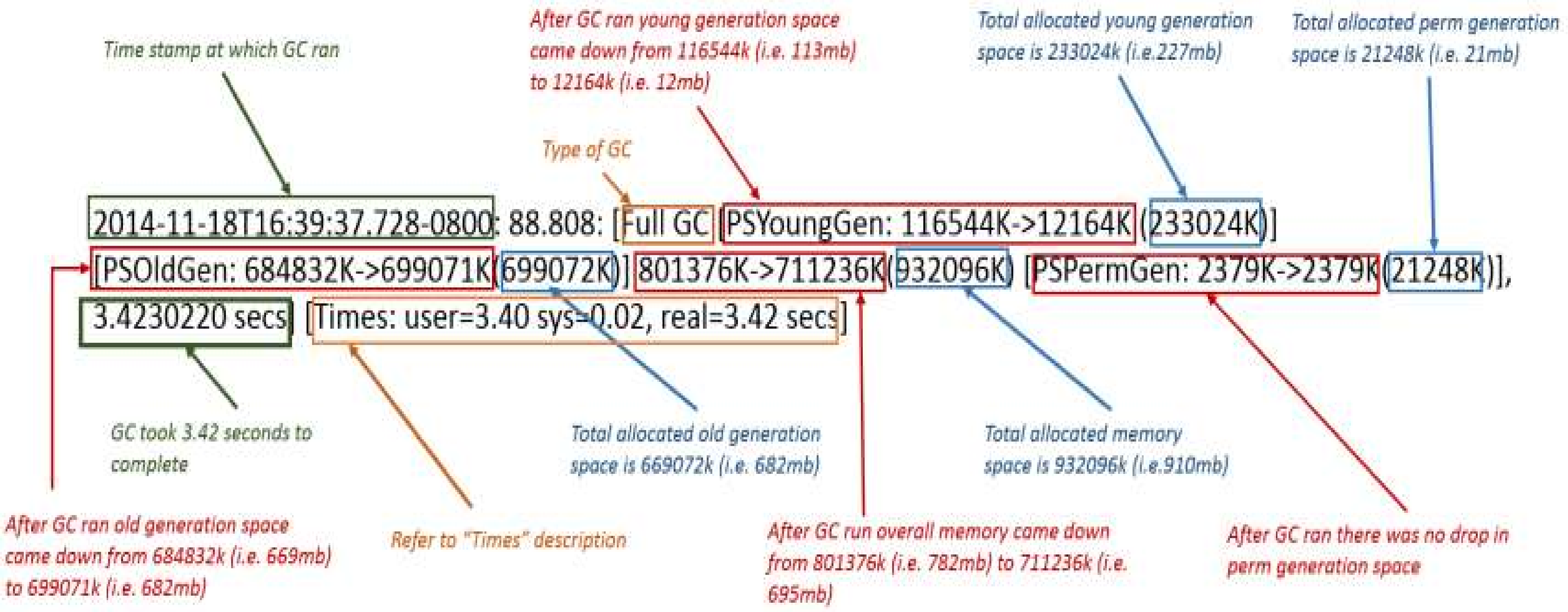
Other JVM flags

-XX:MaxTenuringThreshold	*threshold for moving to old generation*
-XX:+PrintGCDetails	to display the settings and log
-XX:+PrintGCTimeStamps	
-XX:+PrintGCDateStamps	
-Xloggc:file	to save the logs to a file
-verbose:gc	

JVM flags - example

```
-Xms2560m
-Xmx2560m
-XX:NewSize=1536m
-XX:MaxNewSize=1536m
-XX:MetaspaceSize=768m
-XX:MaxMetaspaceSize=768m
-XX:MaxTenuringThreshold=5
-XX:SurvivorRatio=2
```

GC Logs



```
2014-11-18T16:39:37.728-0800: 88.808: [Full GC [PSYoungGen: 116544K->12164K(233024K)]
[PSOldGen: 684832K->699071K(699072K)] 801376K->711236K(932096K) [PSPermGen: 2379K->2379K(21248K)],
3.4230220 secs] [Times: user=3.40 sys=0.02, real=3.42 secs]
```

Time stamp at which GC ran

After GC ran young generation space came down from 116544k (i.e. 113mb) to 12164k (i.e. 12mb)

Total allocated young generation space is 233024k (i.e.227mb)

Total allocated perm generation space is 21248k (i.e. 21mb)

Type of GC

GC took 3.42 seconds to complete

Total allocated old generation space is 669072k (i.e. 682mb)

Total allocated memory space is 932096k (i.e.910mb)

After GC ran old generation space came down from 684832k (i.e. 669mb) to 699071k (i.e. 682mb)

Refer to "Times" description

After GC run overall memory came down from 801376k (i.e. 782mb) to 711236k (i.e. 695mb)

After GC ran there was no drop in perm generation space

How Garbage Collector works

- When an object is no longer used, the garbage collector reclaims the underlying memory and reuses it for future object allocation.
- This means there is no explicit deletion and no memory is given back to the operating system
- Special objects called garbage-collection roots (GC roots) are used to traverse the object graph to identify application-reachable objects
- There are 4 kinds of GC Roots
  - Local variables
  - Active java threads
  - Static variables
  - JNI references

How Garbage Collector works(contd)

GC Roots

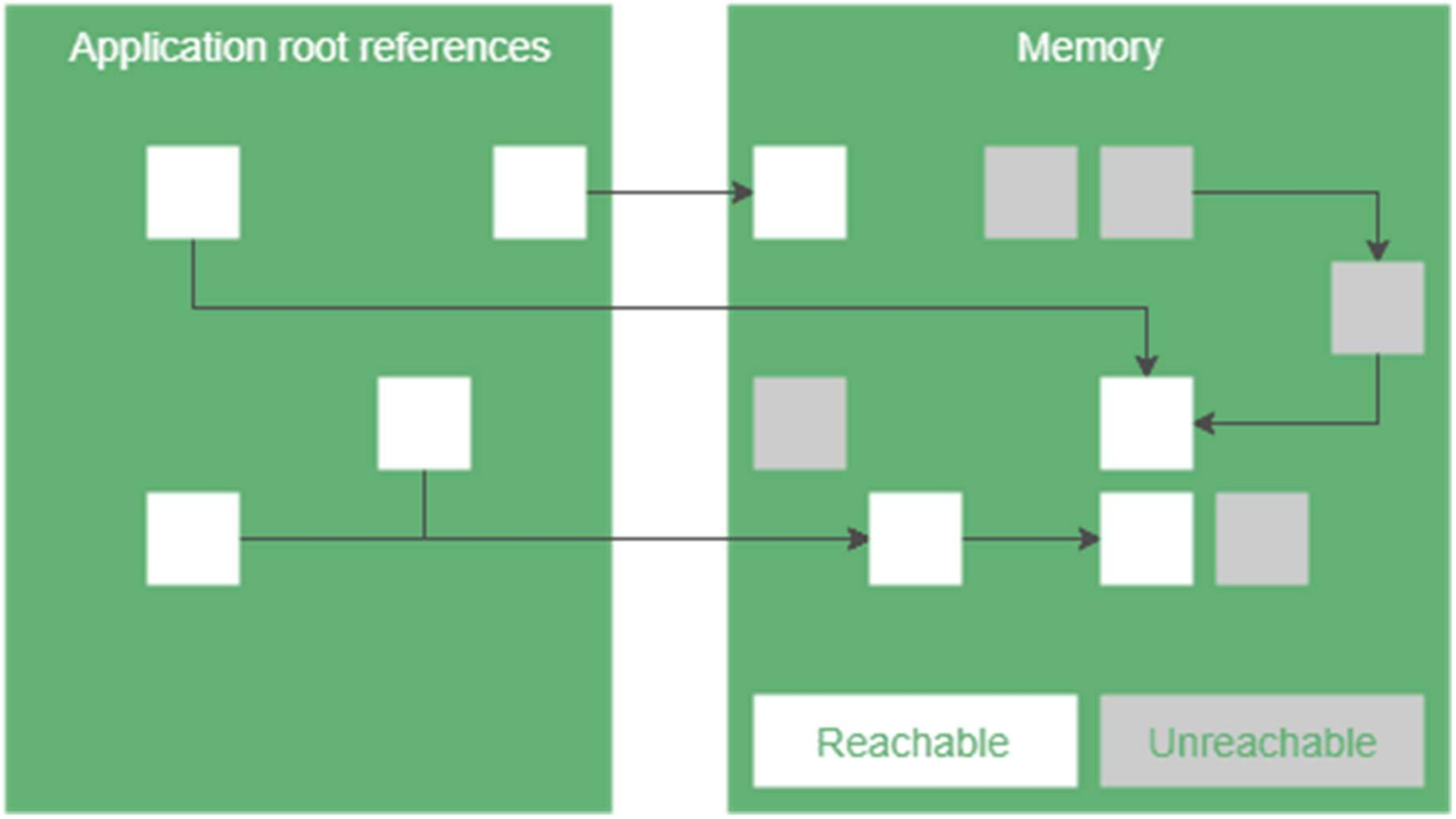
- **Local variables** are kept alive by the stack of a thread.
- For all intents and purposes, local variables are GC roots.

- **Active Java threads** are always considered live objects and are therefore GC roots.
- This is especially important for thread local variables.

- **Static variables** are referenced by their classes and are GC Roots
- Classes themselves can be garbage-collected, which would remove all referenced static variables (App server scenario)

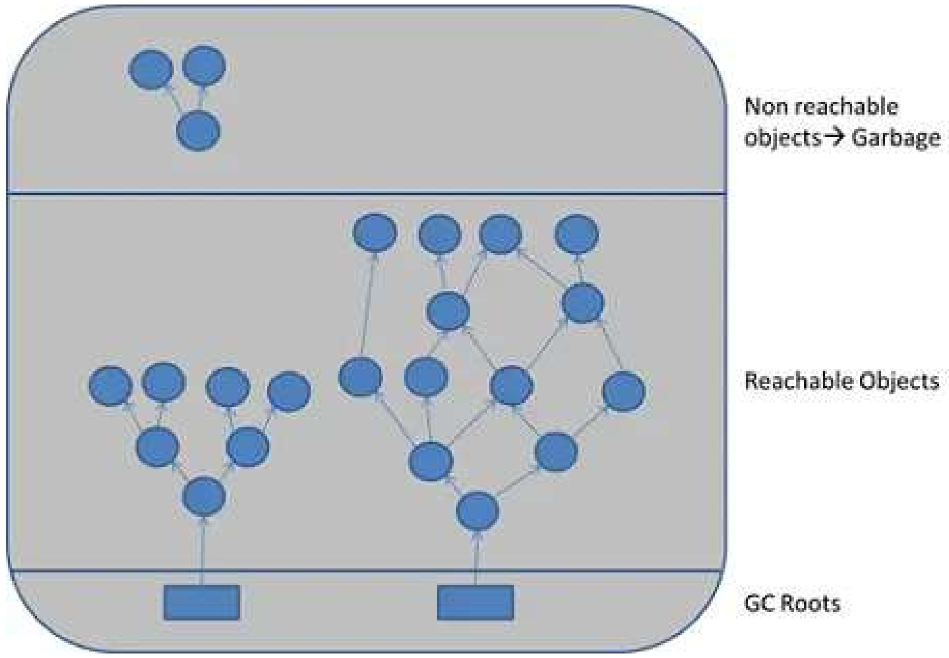
- **JNI References** are Java objects that the native code has created as part of a JNI call
- Such objects represent a very special form of GC root

How Garbage Collector works (contd)



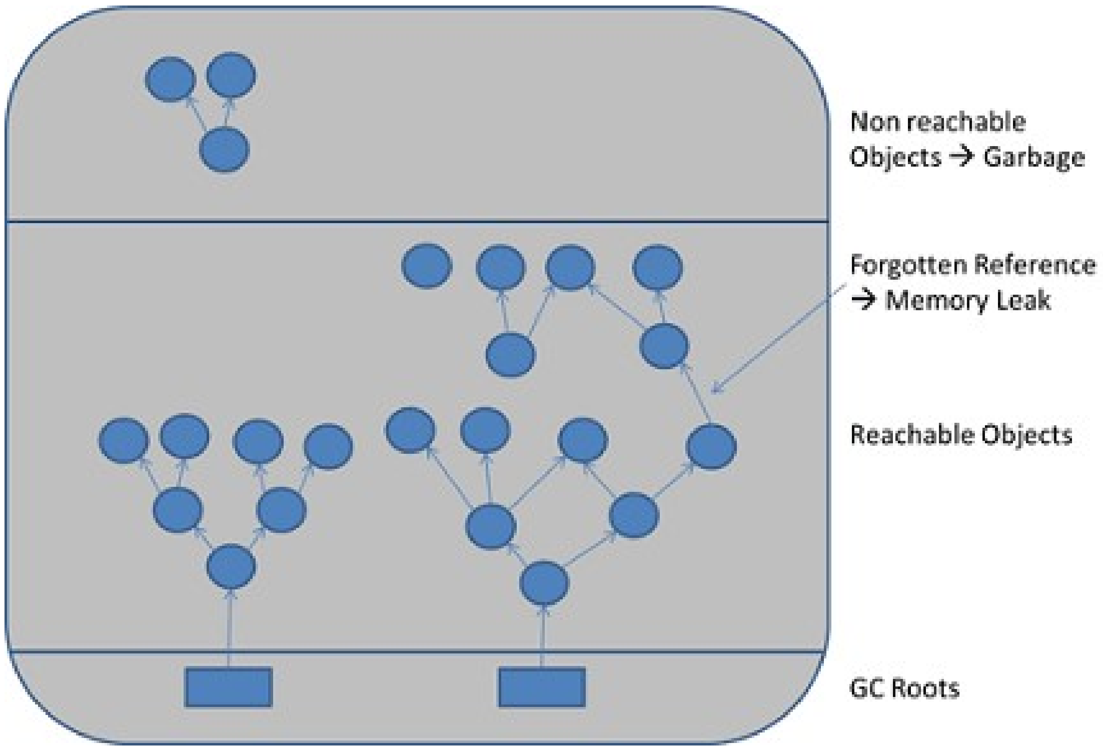
How Garbage Collector works (contd)

- To determine which objects are no longer in use,
- the JVM intermittently runs a mark-and-sweep algorithm
    - The algorithm traverses all object references, starting with the GC roots, and marks every object found as alive
    - All of the heap memory that is not occupied by marked objects is marked as free



How Garbage Collector works (contd)

- It's possible to have unused objects that are still reachable by an application because the developer simply forgot to dereference them
- Such objects cannot be garbage-collected


Non reachable Objects → Garbage
Forgotten Reference → Memory Leak
Reachable Objects
GC Roots

Garbage Collection phases

- **Mark**: garbage collector identifies which objects are in use and which ones are not
- **Sweep**: after the marking phase has completed all space occupied by unvisited objects is considered free and can thus be reused to allocate new objects
- **Compact**: after marking move all marked – and thus alive – objects to the beginning of the memory region
- **Copy:** similar to the Mark and Compact Here the marked objects are copied to different empty memory region.
  - Mark and Copy approach has some advantages as copying can occur simultaneously.
  - The disadvantage is the need for one more memory region, which should be large enough to accommodate survived objects

Types of Garbage Collectors

Serial Garbage Collector

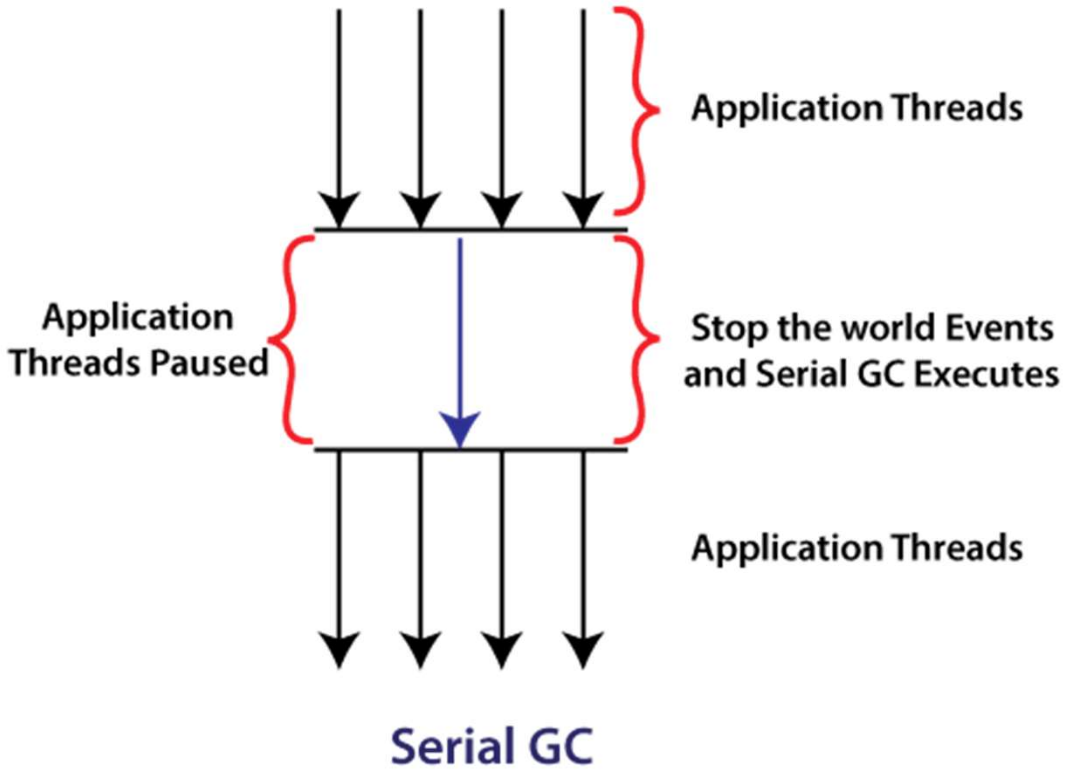
- This is the simplest GC implementation
- it basically works with a single thread
- As a result, this *GC* implementation freezes all application threads when it runs (stop the world)
- It's not a good idea to use it in multi-threaded applications, like server environments.
- It is best suited for applications that don't have small pause time requirements and run on client-style machines

- Flag to select:

  -XX:+UseSerialGC

Types of Garbage Collectors

Serial Garbage Collector


Application Threads
Application Threads Paused
Stop the world Events and Serial GC Executes
Application Threads
Serial GC

Types of Garbage Collectors(contd)

Parallel Garbage Collector

- Parallel Garbage Collector is the **default GC** used by the JVM.
- The working of the parallel garbage collector is the same as the serial garbage collector
- The only difference is that serial garbage collector uses a **single** thread and the parallel garbage collector uses **multiple** threads

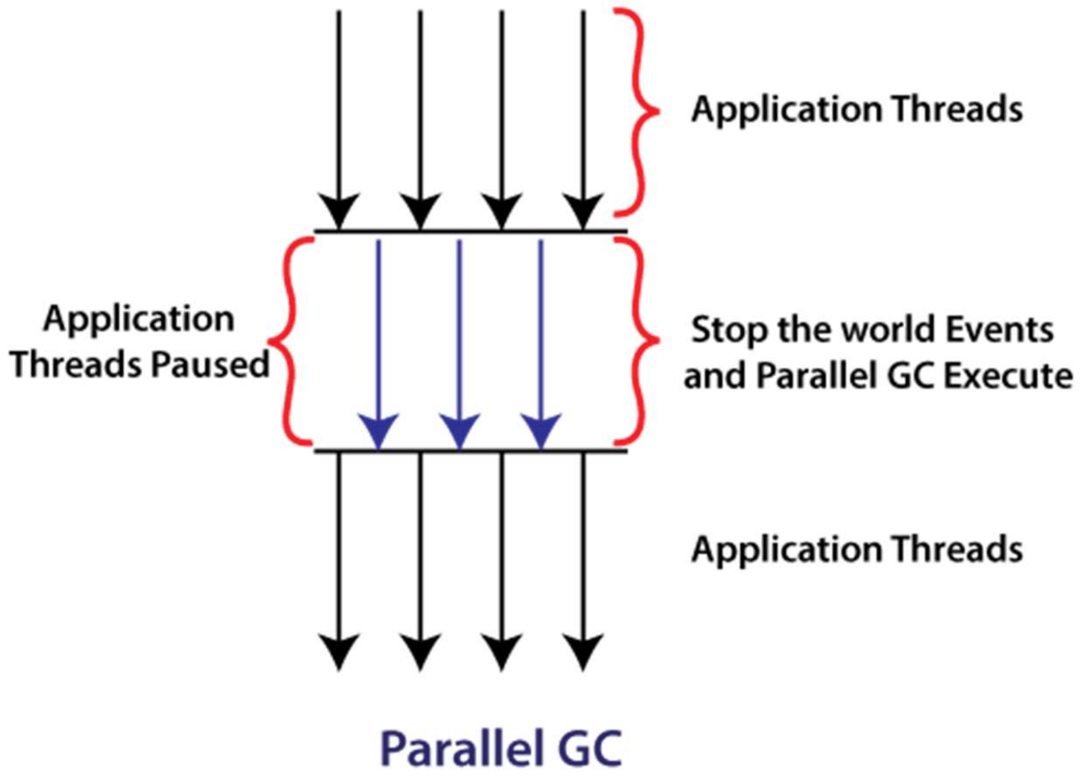
- The numbers of garbage collector threads can be controlled with the JVM option ***-XX:ParallelGCThreads=<N>***

- Flag to select:

  -XX:+UseParallelGC

Types of Garbage Collectors(contd)

Parallel Garbage Collector


Application Threads
Application Threads Paused
Stop the world Events and Parallel GC Execute
Application Threads
Parallel GC

Types of Garbage Collectors(contd)

Parallel Old Garbage Collector

- This is the same as Parallel GC except that it uses multiple threads for both young generation and old generation garbage collection
- Flag to select:

  -XX:+UseParallelOldGC

Types of Garbage Collectors(contd)

Concurrent mark sweep (CMS) Garbage Collector

- CMS uses multiple threads to scan the heap memory consistently to the mark objects and then sweep the marked objects(marked as available space)
- It does not freeze the application's threads during the garbage collection
- GC threads concurrently execute with the application's threads.
- For this reason, it uses more CPU in comparison to other GC
- But it freezes all the threads if any change happens in heap

- Flag to select:

  -XX:+UseConcMarkSweepGC

Types of Garbage Collectors(contd)

G1 (Garbage first) Garbage Collector

- designed for applications running on multi-processor machines with large memory space.
- It’s available from the *JDK7* and in later releases.
- *G1* collector partitions the heap into a set of equal-sized heap regions,
- After the mark phase is complete, *G1* knows which regions are mostly empty
- It collects in these areas first, which usually yields a significant amount of free space

- Flag to select:

  -XX:+UseG1GC

How to make the objects ready for GC

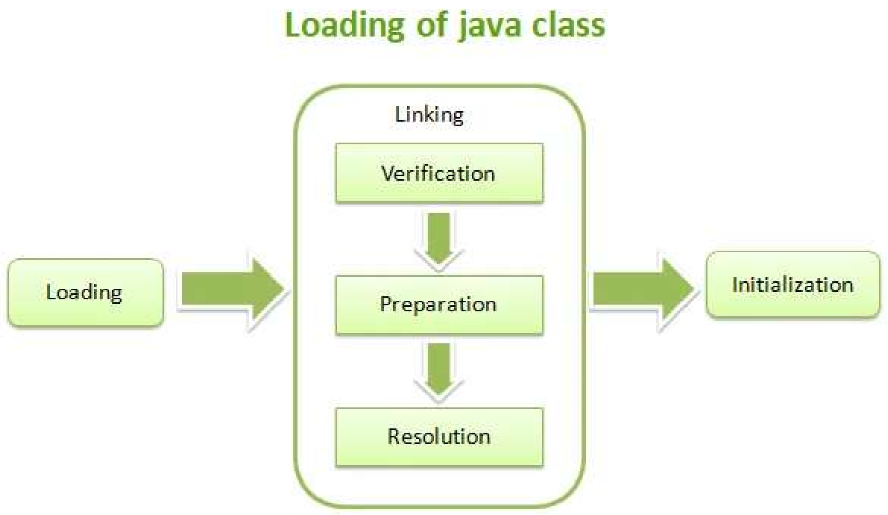
- **Create an object inside a method.** After methods are executed, all objects called within those methods become unreachable
- **Nullify the reference variable.** You can change a reference variable to NULL
- **Re-assign the reference variable.** Instead of nullifying the reference variable, you can also reassign the reference to another object.
- **Create an anonymous object.** An anonymous object doesn’t have a reference, so the garbage collector will mark and remove it during the next garbage collection cycle.

Best practices for GC

- Choose the right garbage collector:
- Monitor and analyze garbage collection logs:
- Optimize heap size:
- Tune garbage collection parameters:
- Minimize object creation:
- Use parallelism and concurrency:

Class Loading Activities

- Java’s dynamic class loading functionality is handled by the Class Loader SubSystem.
- It loads, links and initializes the class when it refers to a class for the first time
- Class Loader SubSystem is responsible for following 3 activities
    1. Loading
    2. Linking
    3. Initialization



Loading

- Loading means reading class files from hard disk and store corresponding binary data in method area.
- For each class file JVM will store corresponding information in method area, such as
    1. Fully qualified name of class
    2. Fully qualified name of immediate parent class
    3. Methods info
    4. Variable info
    5. Constructor info
    6. Modifiers info
    7. Constant pool info
    8. whether .class file represents class or Interface or enum

- After loading .class file JVM creates an object for that loaded class on the Heap memory of type java.lang.Class.

Linking

- Linking consists of three activities

  1. verify (verification)
     - Following points are checked in Verification process.
     - It ensures that Binary representation of a class is a structurally correct or not.
     - JVM will check
       - whether the .class file is generated by valid compiler
       - Whether .class file is properly formatted or not. (bytecode verifier)
     - If the binary representation of a class or interface does not satisfy the static or structural constraints then a VerifyError is thrown

Linking

2. prepare (preparation)
   - In this phase, JVM will allocate memory for class level or interface level static variables and assign default values.

3. resolve (resolution)
   - Resolution is the process of dynamically determining concrete values from symbolic references
   - it is the process of replacing symbolic names (variables) with original memory references from method area

Initialization

- In Initialization phase, all static variables are assigned with original values
- static blocks will be executed from parent to child and from top to bottom.

Illustration

```
public class Test {

    public static void main(String[] arg) {
        String s = new String("Pumpkin");
        Student s1 = new Student();
    }
}
```

- For the above class, class loader loads
  - Test.class
  - Object.class – parent class
  - String.class
  - Student.class

Types of Class Loaders

- Class Loader Subsytem contains following three types of class Loaders

1. Bootstrap Class Loader or primordial Class Loader
2. Extension Class Loader
3. Application Class Loader or System Class Loader

Bootstrap Class Loader

- It is responsible to load core Java API classes i.e. the classes present in rt.jar
- This location is called bootstrap class path i.e. Bootstrap Class Loader is responsible to load classes from bootstrap class path.
- Bootstrap Class Loader is by default available with every JVM.
- It is implemented in native languages like C / C++ and not implemented in java.

Extension Class Loader

- The extension class loader is a child of the bootstrap class loader, and takes care of loading the extensions of the standard core Java classes
- The extension class loader loads from the JDK extensions directory, usually the *$JAVA_HOME/lib/ext* directory

System Class Loader

- It is the child class of Extension Class Loader
- This class loader is responsible to load classes from application class path
- It internally uses environment variable classpath
- It is implemented in java and the corresponding .class file name is sun.misc.Launcher$AppClassLoader.class

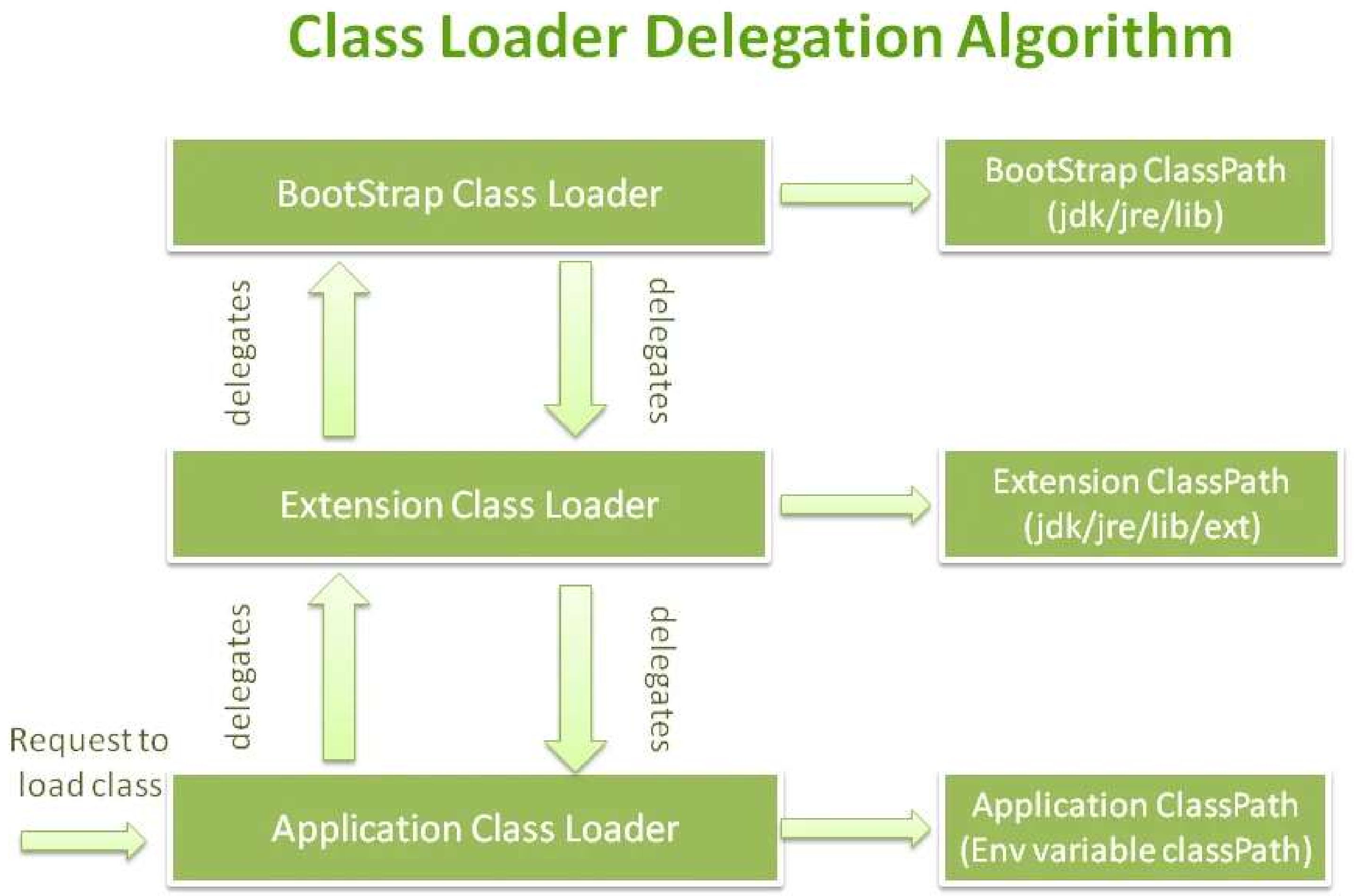
Class Loading Mechanism

- Whenever JVM come across particular class, first it will check whether the corresponding .class file is loaded or not.
- If it is already loaded in method area, then JVM will uses it
- If it is not loaded then JVM request class loader sub system to load that particular class.
- Then class loader subsystem handsover the request to Application class loader.
- Application class loader delegates the request to Extension class loader
- Extension class loader in turn delegates the request to Bootstrap class loader.

Class Loading Mechanism

- Then Bootstrap class loader will search in Bootstrap class path and load it
- If it is not available Bootstrap class loader delegates the request to Extension class loader.
- Extension class loader will search in Extension class path and load the class
- If not available, extension class loader delegates the request to application class loader.
- Application class loader will search in Application class path.
- If it is available, it will be loaded otherwise we will get runtime exception saying ClassNotFoundException.

Class Loading Mechanism (contd)


Class Loader Delegation Algorithm
BootStrap Class Loader
BootStrap ClassPath (jdk/jre/lib)
delegates
delegates
Extension Class Loader
Extension ClassPath (jdk/jre/lib/ext)
delegates
delegates
Request to load class
Application Class Loader
Application ClassPath (Env variable classPath)

Advantages

- As a consequence of the delegation model, it's easy to ensure unique classes, as we always try to delegate upwards.
- For example if we create java.lan.String, it is never loaded

- In addition, classes loaded by child class loader can use classes loaded by parent class loader
- For instance, classes loaded by the system class loader can use classes loaded by the extension and bootstrap class loaders, but not vice-versa
- To illustrate this, if Class A is loaded by the application class loader, and class B is loaded by the extensions class loader, then both A and B classes are visible as far as other classes loaded by the application class loader are concerned.
- Class B, however, is the only class visible to other classes loaded by the extension class loader.

CustomClassLoaders

- Custom class loaders are helpful for more than just loading the class during runtime.
- A few use cases might include:
    1. Helping to modify the existing bytecode
    2. Creating classes dynamically suited to the user’s needs, e.g. in JDBC, switching between different driver implementations is done through dynamic class loading.
    3. Implementing a class versioning mechanism while loading different bytecodes for classes with the same names and packages
    4. Getting class code from network

CustomClassLoaders

- **Isolation**: You want to load and unload plugins dynamically without affecting the main application. Custom class loaders enable isolation by loading each plugin in its own class loader.

- **Versioning**: If you have different versions of a library in your application, custom class loaders can help you manage class versioning, ensuring that each component uses the appropriate version

- **Dynamic Loading**: For situations where you want to load classes from sources other than JAR files, such as databases or remote services, custom class loaders are invaluable.

CustomClassLoaders

```
public class CustomClassLoader extends ClassLoader {

    @Override
    public Class findClass(String name)  {
        byte[] b = loadClassFromFile(name);
        return defineClass(name, b, 0, b.length);
    }

    private byte[] loadClassFromFile(String fileName)  {

    }
}
```

Profilers

- A Java Profiler is a tool that monitors operations at the JVM level
- These code constructs and operations include object creation, thread executions, memory usage and garbage collections

- Some known profilers:
  - Jprofiler
  - Yourkit
  - Java VisualVM
  - the NetBeans Profiler
  - the IntelliJ Profiler.

Memory Leaks

Class loaded multiple times

- It is the very purpose of a classloader to load classes in isolation to each other
- Application servers use this feature of classloaders to load different applications or parts of applications in isolation
- This makes it possible to load multiple versions of the same library for different applications.
- Due to configuration errors, we can easily load the same version of a library multiple times

- Make sure you do not have too many class loaders

Class loader leaks

- As classes are referenced by their classloaders,
- they get removed when the classloader is garbage-collected.
- That will happen only when the application gets unloaded
- A classloader will be removed by the garbage collector only if nothing else refers to it.
- All classes hold a reference to their classloader and all objects hold references to their classes.
- As a result, if an application gets unloaded but one of its objects is still being held (e.g., by a cache or a thread-local variable), the underlying classloader is not removed by the garbage collector
- This will happen only if you redeploy your application without restarting the application server

Mutable static fields

- Static fields are de facto GC roots, which means they are never garbage-collected!
- For convenience alone, static fields and collections are often used to hold caches or share state across threads.
- Mutable static fields need to be cleaned up explicitly.
- The cleanup will not take place, resulting in a memory leak.

- Never use mutable static fields—use only constants.
- If you need mutable static fields, make sure of tracking them or try some other technique

JNI Memory Leaks

- Java Native Interface (JNI) memory leaks are hard to find.
- Every Java object created in a native method begins its life as a local reference,
- which means that the object is referenced until the native method returns.
- In some cases you want to keep the created object even after the native call has ended.
- To achieve this you can either ensure that it is referenced by some other Java object or you can change the local reference into a global reference
- The only way to discover JNI memory leaks is to use a heap-dump tool that explicitly marks native references.
- It's better to assign the desired object to the field of a normal Java class.

ThreadLocal Variables

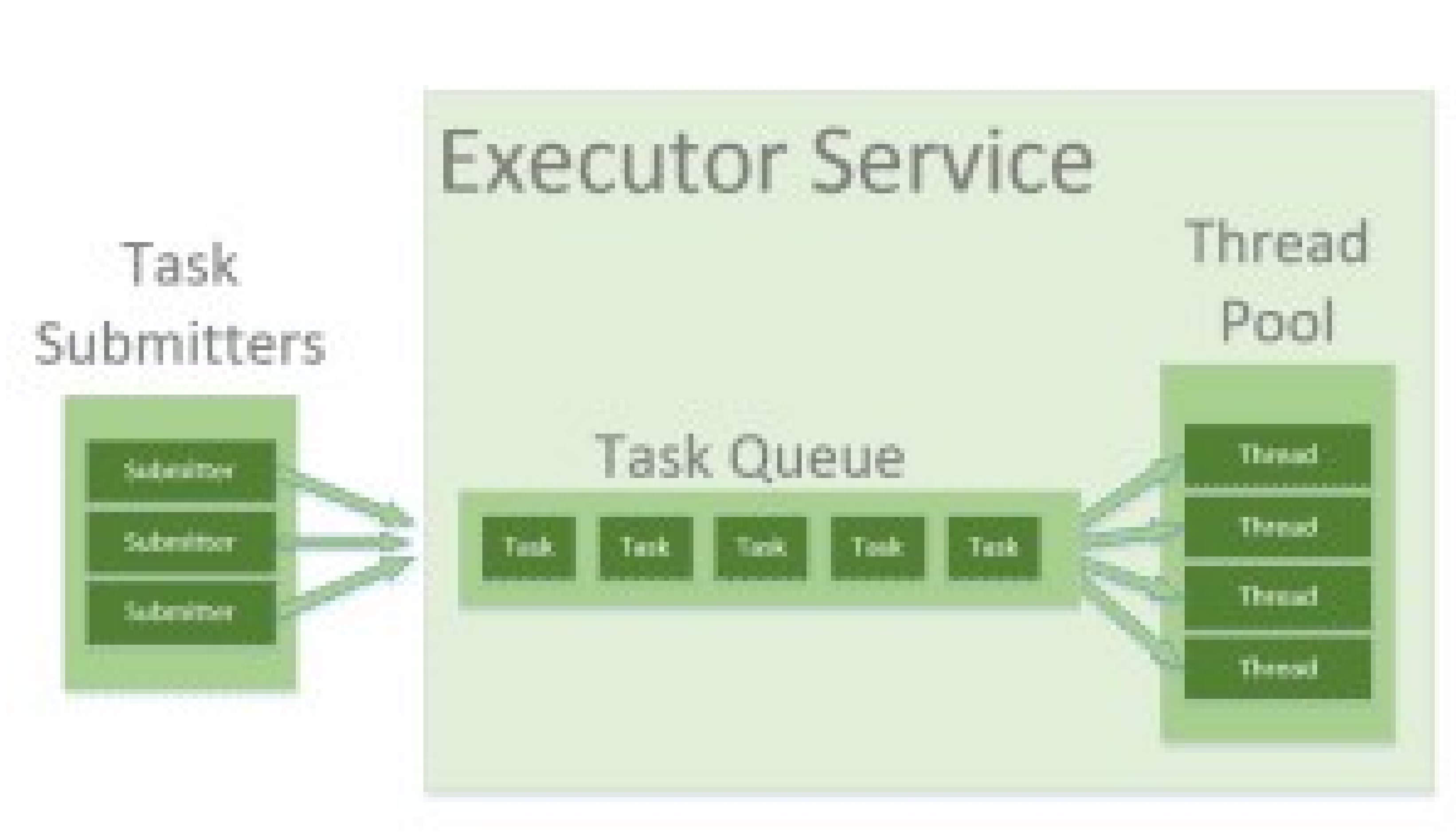
- *ThreadLocal* is a feature that enables the creation of thread-specific variables.
- Unlike regular variables that are shared across threads, a *ThreadLocal* variable provides each thread with its own, independent copy.
- This is particularly useful in multi-threading environments where data isolation per thread is required

- *ThreadLocal* initializes the variable the first time a thread accesses it, using an *initialValue* method.
- The thread can then set or get this variable’s value throughout its execution

Thread Pool

- The Thread Pool pattern helps to save resources in a multithreaded application and to contain the parallelism in certain predefined limits
- When we use a thread pool, we write our concurrent code in the form of tasks and submit them for execution to an instance of a thread pool.
- This instance controls several re-used threads for executing these tasks

Thread Pool



Executor Service
Task Submitters
Submitter
Submitter
Submitter
Task Queue
Task
Task
Task
Task
Task
Thread Pool
Thread
Thread
Thread
Thread

ThreadLocal in Thread Pool

- *the application borrows a thread from the pool.*
- *Then it stores some thread-confined values into the current thread's ThreadLocal.*
- *Once the current execution finishes, the application returns the borrowed thread to the pool.*
- *After a while, the application borrows the same thread to process another request.*
- *Since the application didn't perform the necessary cleanups last time, it may re-use the same ThreadLocal data for the new request.*
- *This may cause surprising consequences in highly concurrent applications.*

ThreadLocal Memoy Leaks

- *ThreadLocals* are supposed to be garbage collected once the holding thread is no longer alive.
- But the problem arises when we use *ThreadLocals* in Thread pools
- Modern application servers use a pool of threads to process requests, instead of creating new ones
- If any class creates a *ThreadLocal* variable, but doesn't explicitly remove it, then a copy of that object will remain with the worker *Thread*

- *ThreadLocals* provide the ***remove()*** method, which removes the current thread's value for this variable

Problem with large classes

- At times it is possible to have a class with lot of static variables and methods
- These classes occupy lot of memory
- Split these classes into smaller classes so that only those classes which are being used remain in the memory and the other classes are unloaded when not used

Coding Techniques

Understanding Collections

Interface	Class	Synchronized?	
Set	HashSet	No	Fastest Set; slower than HashMap but implements the Set interface (HashMap does not)
	TreeSet	No	Slower than HashSet; provides iteration of keys in order
Map	HashMap	No	Fastest Map
	Hashtable	Yes	Slower than HashMap, but faster than synchronized HashMap
	TreeMap	No	Slower than Hashtable and HashMap; provides iteration of keys in order
List	ArrayList	No	Fastest List

Optimize the code

- Understand String class
- Understand StringBuilder and StringBuffer classes
- Use regex for string complex comparisons
- Avoid BigInteger and BigDecimal
- Avoid large methods
- Avoid large classes (split them)
- Understand primitives vs Wrappers
- Database connection pooling
- Use PrearedStatement instead of Statement in JDBC

JDBC Statement

- The JDBC Statement class has a few flaws

```
Statement st = con.createStatement();
String username = “ramana"; // user input
String query = "SELECT user FROM users WHERE name = '" + username + "'";
ResultSet result = st.executeQuery(query);
```

- If the user were to pass ‘ OR 1=1– the resulting query would become:

```
SELECT user FROM users WHERE username = '' OR 1=1 -- ‘
```

- This would cause the query to return all records since the OR 1=1 condition is always true.
- The double dash — at the end is a comment in SQL, which causes the rest of the original query to be ignored.
- This type of attack is called SQL Injection.

Logging

- Before you create a debug message, you should always check the current log level first.
- Otherwise, you might create a *String* with your log message that will be ignored afterward

- The following code builds a string which may not be used

```
log.debug(String.format("User [%s] called method X with [%d]", userName, i));
```

- It's better to check the current log level first before you create the debug message.

```
if (log.isDebugEnabled()) {
log.debug("User [" + userName + "] called method X with [" + i + "]");
}
```

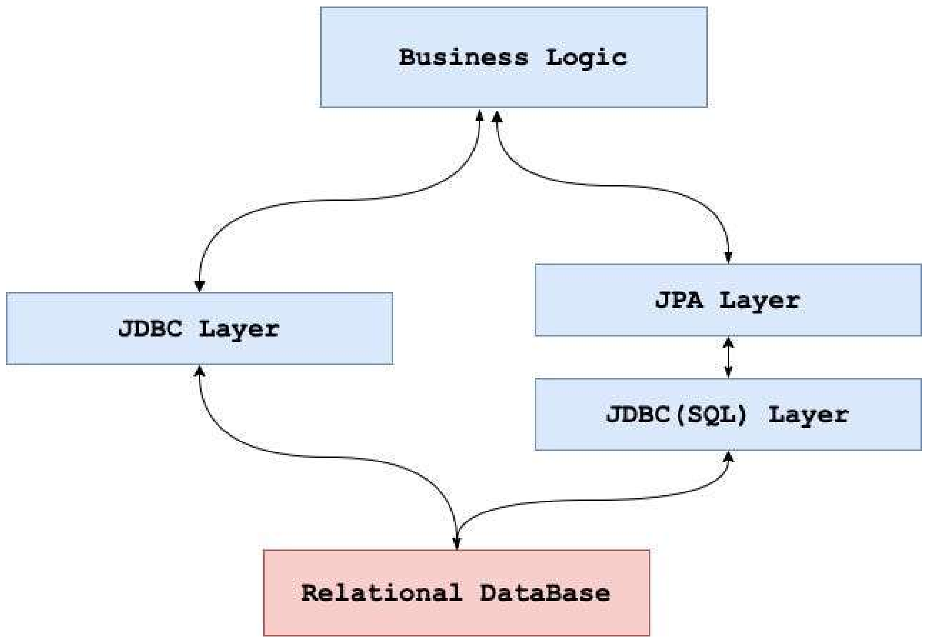
Recursion

- Recursion is good in solving complex problems where loops cannot be used
- However, you should use recursion sparingly
- In the iterative approach, the local variables are created once.
- However, in the case of the recursion, for each method call a stack frame with local variables is created.

- The rule is – do not use recursion if loops can be used

Is JPA preferable to JDBC

- JPA is a Java standard for binding Java objects to records in a relational database
- Handling object persistence is breeze with JPA
- But for simple operations JPA has its own overhead



Optimize SQL queries

- Add missing indexes
- Use indexes effectively
- Check for unused indexes
- Reduce the use of wildcard characters
- Use wildcards at the end of a phrase only
- Use appropriate data types and layouts
- Avoid redundant or unnecessary data retrieval
- use exists() instead of count()
- Avoid subqueries (join may be better)
- Avoid too many JOINs
- Avoid using SELECT DISTINCT
- Use SELECT fields instead of SELECT *
- Minimize large write operations
- Create joins with INNER JOIN (not WHERE)